

# Avaliação e Trabalhos de Laboratório de Compiladores

José de Oliveira Guimarães  
Departamento de Computação  
UFSCar - Sorocaba, SP  
Brasil

e-mail: josedoliveiraguimaraes@gmail.com

August 22, 2019

A avaliação da disciplina 1001308 - Laboratório de Compiladores (quatro créditos) consiste de duas provas (NP1 e NP2) e dois trabalhos (NT1 e NT2). A nota final NF é calculada como

```
se N1 > 7 e N2 < 6
    NF = (N1 + 3*N2)/4
senão
    NF = (N1 + N2)/2
```

sendo Ni (para i = 1, 2) calculados como

```
se NTi >= 6 e NPi >= 6
    então
        Ni = (NTi + NPi)/2
senão
    Ni = menor entre NPi e NTi
```

A avaliação da disciplina 480665 - Laboratório de Compiladores (dois créditos) consiste de uma prova (NP) e um trabalho (NT). A nota final NF é calculada como

```
se NT >= 6 e NP >= 6
    então
        NF = (NT + NP)/2
senão
    NF = menor entre NP e NT
```

A descrição dos trabalhos se segue. Os trabalhos podem ser individuais ou em grupos de dois. Os alunos da disciplina 480665 deverão fazer apenas o primeiro trabalho.

A nota de cada trabalho será dividida em duas: 8 pontos para o trabalho entregue na data final do trabalho e 2 pontos para uma ou mais notas intermediárias.

## 1 Primeiro Trabalho

O primeiro trabalho consiste da construção de:

1. um analisador sintático e semântico da linguagem Cianeto. Esta linguagem é descrita no texto “The Cianeto Language”. Não é necessário implementar campos e métodos *shared*;
2. um gerador de código de Cianeto para Java utilizando métodos da Árvore de Sintaxe Abstrata (ASA). Deve haver um método

```
public void genJava(PW pw)
```

na classe `Program` da ASA (já está assim no compilador fornecido pelo professor). E métodos com este mesmo nome nas outras classes da ASA. A classe principal do arquivo Java, a única pública, deve ter o mesmo nome que o arquivo. Se o nome do arquivo é “OK\_GER01.ci”, deve haver a seguinte classe no arquivo Java gerado:

```
public class OK_GER01 {
    public static void main(String []args) {
        new Program().run();
    }
}
```

Para testar a geração de código em Java de um arquivo

`C:\Dropbox\OK_GER01.ci`

chame o seu compilador com os parâmetros

```
"C:\Dropbox\OK_GER01.ci" -genjava "C:\Dropbox"
```

O arquivo Java produzido será colocado em `C:\Dropbox`, compilado e executado.

Os arquivos que testam se o código foi gerado corretamente tem nomes `OK_GERxx.ci` no qual `xx` é 01, 02 etc. Estes arquivos foram feitos de tal forma que, ao serem executados, a primeira linha que eles produzem como saída é igual ao restante da saída. A menos de espaços e linhas em branco. Isto é, se a saída for

```
4 1 2 3 4
4
1 2
3 4
```

então isto quer dizer que o compilador produziu código correto para este arquivo. Mas se a saída for

```
4 1 2 3 4
41234
```

então o código produzido está incorreto — a saída não possui os espaços que existem na primeira linha.

Quando o compilador de Cianeto for usado com a opção `-genjava`, ele irá chamar o método `genJava` do programa, gerar um arquivo Java, compilá-lo (com `javac` e interpretá-lo (com o interpretador de Java). E irá, pela saída do programa, verificar se ele está gerando código corretamente. Um relatório chamado “`report.txt`” será produzido no diretório que se segue à opção `-genjava`. Este relatório conterà todos listas com arquivos que não compilaram, que geraram código incorreto etc.

Data de entrega do primeiro trabalho: 07/out para alunos de 1001308 (quatro créditos) e 02/dez para alunos de 480665 (dois créditos). A prova correspondente a este trabalho será aplicada no dia 02/dez.

## 2 Segundo Trabalho

Faça a geração de código de Cianeto para C. Siga precisamente as instruções do texto “Geração de Código em C para Cianeto”.

Data da prova e entrega do segundo trabalho: 02/dez.

## 3 Outras Observações

Os que ficarem com nota final abaixo de 6 terão a oportunidade de fazer uma recuperação em dezembro. Esta recuperação consiste do mesmo trabalho mais uma nova prova. Para os alunos de 1001308, será aplicada uma única prova com todo o conteúdo das duas provas anteriores. Sendo PR a nota da prova, TR a nota do trabalho e ANF a nota final anterior, a nova nota final NNF será calculada como

```
se TR >= 6 e PR >= 6
então
    NNF = 0,2*ANF + 0,8*(TR + PR)/2
senão
    NNF = 0,2*ANF + 0,8*(menor entre PR e TR)
```

A data da nova prova e entrega do trabalho é 09/dez.

Para cada trabalho, submeta ao AVA um único arquivo zip cujo nome é Nome1-Nome2.zip, sendo Nome1 e Nome2 os nomes completos dos integrantes do grupo, em ordem alfabética. Não use acentos nos nomes, omita algum sobrenome se achar necessário. O conteúdo do arquivo Nome1-Nome2.zip deve ser um diretório 'src' com o código fonte do compilador. Coloque apenas os arquivos \*.java nos diretórios apropriados. Em resumo, não envie nenhum arquivo \*.class, \*.txt, testes, executáveis etc.

Use a classe Comp.java que está no AVA sem **nenhuma** alteração. Esta é a classe principal do compilador, que chama o método `compile` da classe `Compiler`. Para chamar o compilador, digite algo como

```
C:\Dropbox\19-2\LabComp\cianeto\bin>java -cp .
    comp.Comp "C:\Dropbox\19-2\LC\tests"
```

O compilador, quando chamado com um diretório como argumento, compila todos os arquivos \*.ci do diretório e produz um relatório chamado "report.txt". O compilador também pode ser chamado com um único arquivo.

Até 5% de código do compilador pode ser copiado de outros grupos. Mas é necessário especificar exatamente qual trecho e de qual grupo ele foi copiado em um arquivo "trechos-copiados.txt". Este arquivo deve estar no diretório principal do arquivo zip com o compilador. Estes 5% se referem aos trechos feitos por você, não ao total do compilador, parte dele foi fornecida no AVA. Importante: **os métodos de análise sintática e semântica de expressões NÃO podem ser copiados**. Isto é, os métodos que fazem a análise da regra `Expression` da gramática não podem ser copiados. Obviamente inclui todos os métodos que analisam expressões.

O seu compilador deve ter obrigatoriamente as características descritas abaixo.

- (1) O trabalho deve ser feito em Java e a classe principal deve ser a fornecida no AVA. Você pode usar ou não o restante do compilador fornecido pelo professor. Mas é imprescindível emitir os erros usando os métodos que já estão prontos, que usam as classes `Compiler`, `CompilerError` etc.

- (2) **Todos os arquivos devem ter um comentário inicial com o nome dos integrantes do grupo.** Todos os arquivos (repetindo). Sem estes comentários, o trabalho não será considerado.
- (3) Use codificação Cp1252 para os arquivos, que é o geralmente usado no IDE Eclipse para Windows. Muitos trabalhos feitos nos computadores da Apple não respeitam esta restrição. Então converta os arquivos para Cp1252 antes de comprimi-los e colocá-los no AVA. Se você usar um editor convencional, os arquivos já estarão em Cp1252 ou em uma codificação que não causa erros de compilação.
- (4) **NÃO** copie regras da gramática do pdf “The Cianeto Language” e cole-as no seu código fonte (em geral, no arquivo “Compiler.java”). Isto causa erros de compilação quando se usa o compilador `javac` da linha de comando. Você pode perder pontos por fazer isto.

Você pode fazer perguntas por email nesta disciplina, excepcionalmente.

## Apêndices

### A O Relatório do Compilador

Chame o compilador com o diretório de testes. Ele produzirá um relatório “`report.txt`” no diretório corrente. Este arquivo inicia-se com um resumo das compilações dos arquivos do diretório. Algo assim:

```
-----
MI:  25          I:  11          PI:  29          Exc:  9
Dev: 72/130/55%  LE: 15/130/11%   SSE: 24/70/34%
```

MI = muito importante, I = importante, PI = pouco importante, Exc = exceções  
Dev = deveria ter sinalizado, LE = sinalizou linha errada, SSE = sinalizado sem erro

```
-----
```

MI é o número de testes em que o compilador falhou e que são considerados muito importantes.  
I é o número de testes em que o compilador falhou e que são considerados importantes.  
PI é o número de testes em que o compilador falhou e que são considerados pouco importantes (mas são importantes!).  
Dev é seguido de três números. O primeiro é o número de testes em que o compilador deveria ter sinalizado erro mas não o fez (se o compilador está correto, este número deve ser 0). O segundo é o número de testes em que há erro (há uma anotação `cep` em cada um deles). O terceiro número é a porcentagem do primeiro número em relação ao segundo.  
LE é seguido de três números. O primeiro é o número de testes em que o compilador apontou o erro, mas na linha errada. O segundo é o número de testes em que há erro (igual ao número de Dev). O terceiro número é a porcentagem do primeiro número em relação ao segundo.  
SSE é seguido de três números. O primeiro é o número de testes em que o compilador sinalizou erros mas que não possuem erros (em um compilador correto, este primeiro número deve ser 0). O segundo é o número de testes em que **não** há erro (um arquivo sem erro possui uma anotação `nce`). Então este é o número de arquivos com anotação `nce`). O terceiro número é a porcentagem do primeiro número em relação ao segundo.

## B Dicas Sobre o Trabalho

A classe que representa uma variável local pode ser, inicialmente,

```
public class Variable {
    private String name;
    private Type type;
}
```

Como `type` é do tipo `Type`, este campo (variável de instância) pode apontar para objetos de `Type` e suas subclasses, o que inclui `TypeCianetoClass`. Assim, o tipo de uma variável pode ser “`Int`” (objeto de `Type`), “`Boolean`” (objeto de `Type`), “`String`” (objeto de `Type`) ou uma classe (objeto de `TypeCianetoClass`). Naturalmente, `TypeCianetoClass` deve herdar de `Type` para que isto seja possível.

O construtor de `TypeCianetoClass` deve ter um único parâmetro, o nome da classe. Assim, pode-se criar um objeto de `TypeCianetoClass` tão logo saibamos o nome da classe. Isto é necessário pois o objeto que representa a classe deve logo ser inserido na Tabela de Símbolos, pois uma classe pode declarar um objeto dela mesma:

```
class A
    var A x
    ...
end
```

Assim, ao encontrar o “`x`”, haverá uma busca na tabela de símbolos e lá será encontrado o objeto de `TypeCianetoClass` que representa a classe `A`, que foi inserido lá tão logo o nome da classe se tornou disponível. A mesma observação vale para a classe que representa um método, podemos ter chamadas recursivas.

Ao encontrar um comando

```
x = y;
```

o compilador deve procurar por `x` na tabela de símbolos de tal forma que o objeto de

`AssignmentStatement`<sup>1</sup>

correspondente a esta atribuição tenha um ponteiro para o objeto `Variable` representando `x`. Supõe-se que `Variable` é a classe que representa uma variável (local, campo, ou parâmetro, sendo que deve haver subclasses de `Variable` para cada uma destas categorias). Este objeto é o que foi criado na declaração de `x`. O mesmo se aplica a `y`. Você deve fazer algum assim:

```
// lexer.getStringValue() retorna "x"
Variable left = symbolTable.searchVariable( lexer.getStringValue() );

if ( left == null ) { error.show("..."); }

return new AssignmentStatement( left, expr() );
```

Este código assume que existe um método `searchLocalVariable`, que pesquisa por uma variável local (inclusive parâmetros), na tabela de símbolos.

Duas estratégias ERRADAS são dadas abaixo.

---

<sup>1</sup>Suponha que esta seja a classe que representa uma atribuição. Ela não foi fornecida, crie algo assim, não necessariamente com este nome.

1. representar `x` como `String`. A classe `AssignmentStatement` seria

```
class AssignmentStatement {
    private String    leftSide;
    private Expr      rightSide;
    ...
}
```

2. representar `x` como uma variável, mas criar esta variável ao encontrar `x`:

```
// lexer.getStringValue() retorna "x"

Variable left = new Variable( lexer.getStringValue() );

return new AssignmentStatement( left, expr() );
```

Os únicos lugares onde deve-se criar objetos representando variáveis locais, campos (variáveis de instância) e parâmetros é na declaração das variáveis correspondentes. E nunca se deve representar variáveis por strings — utilize objetos de `Variable` (não necessariamente com este nome) e suas subclasses. O mesmo se aplica a `TypeCianetoClass` e ao tipo de variáveis. Isto é, um tipo deve ser um ponteiro para um objeto de `Type` e subclasses, não uma string.

Os compiladores sinalizam uma exceção depois de mostrar um erro. Esta exceção deve ser (é) capturada em um bloco `try` no método `compile` da classe `Compiler`. A impressão da pilha de chamadas (com `e.printStackTrace()`) deve ser utilizada apenas na fase de depuração do compilador e não deve estar presente no compilador entregue ao professor

É quase obrigatório que alguma variável referencie o objeto que representa a classe atualmente sendo compilada. Esta variável deve ser um campo de `Compiler` e será largamente usada. Por exemplo, para inserir métodos e campos na classe corrente de `Cianeto`. Idem para o método atualmente sendo compilado.

Utilize a classe `PW` para fazer a tabulação do código gerado corretamente. Esta classe está no pacote `ast`. O construtor toma um objeto `PrintWriter` utilizado para saída

```
pw = new PW(out);
```

sendo `out` um objeto de `PrintWriter`. Na verdade, isto já é feito pelo compilador. Você nunca precisará criar um objeto `PW` ou criar um arquivo. Utilize a infraestrutura fornecida com o compilador.

`pw` possui métodos `printIdent` e `printlnIdent` que automaticamente indentam o que você imprime com `pw.printIdent` ou `pw.printlnIdent`. Naturalmente, `printlnIdent` imprime a string e pula para a próxima linha, enquanto que `printIdent` não. Se você quiser aumentar a indentação, utilize “`pw.add()`”. Para diminuir, utilize “`pw.sub()`”. Teste o seguinte código

```
PrintWriter out = new PrintWriter( ... );
/* faça isso uma única vez antes do início da geração de código,
   o que já é feito pelo compilador fornecido. */
pw = new PW(out);

pw.add();
pw.printIdent("a = a + 1;");
pw.add();
```

```

pw.printIdent("if");
pw.println( "  a > b");
// código sem indentação, pois foi escrito com pw.print
// e não pw.printIdent
pw.printIdent("then");
pw.add();    // comandos dentro do then devem ser indentados
pw.printlnIdent("a = b;");
pw.sub();    // diminui a indentação: acabou o then
pw.printlnIdent("endif");
pw.sub();
pw.println("Texto normal, não indentado");
pw.printlnIdent("Indentado");
pw.sub();

```

A saída deste código é

```

a = a + 1;
if a > b
then
    a = b;
endif
Texto normal, não indentado
Indentado

```